Roboconf: a Hybrid Cloud Orchestrator to Deploy Complex Applications

Linh Manh Pham, Alain Tchana, Didier Donsez, Noel de Palma, Vincent Zurczak, Pierre-Yves Gibello

Roboconf: a Hybrid Cloud Orchestrator to Deploy Complex Applications

Linh Manh Pham[1], Alain Tchana[2], Didier Donsez[1], Noel de Palma[1], Vincent Zurczak[3], Pierre-Yves Gibello[3]
[1]University of Grenoble Alpes, Grenoble, France. E-mail: first.last@imag.fr
[2]University of Toulouse, Toulouse, France. E-mail: alain.tchana@enseeiht.fr
[3]Linagora, Grenoble, France. E-mail: (vincent.zurczak, pygibello)@linagora.com

***Abstract*—This paper presents Roboconf, an open-source distributed application orchestration framework for multi-cloud platforms, designed to solve challenges of current Autonomic Computing Systems in the era of Cloud computing. It provides a Domain Specific Language (DSL) which allows to describe applications and their execution environments (cloud platforms) in a hierarchical way in order to provide a fine-grained management. Roboconf implements an asynchronous and parallel deployment protocol which accelerates and makes resilient the deployment process. Intensive experiments with different type of applications over different cloud models (e.g. private, hybrid, and multi-cloud) validate the genericity of Roboconf. These experiments also demonstrate its efficiency comparing to existing frameworks such as RightScale, Scalr, and Cloudify.**

***Index Terms*—Cloud; Autonomic computing; Installation**

I. Introduction

In the early 2000s, IBM [1] proposed to automate the administration of complex applications throughout the use of what we called Autonomic Computing Systems (ACS for short). The complex applications may contain a lot of components distributed on various tiers and consume large amounts of resources located at multiple sites. This practice consists in transferring human administration knowledge and behaviours to an ACS. As summarized by [1], administration tasks can be divided into the following categories: installation or initial provisioning, reconfiguration and uninstallation. ACSs have proved their usefulness and now the major part of research in this topic focuses on reconfigurations tasks [2].

However, recent years have seen the development and now the democratization of a new technology called Cloud computing which is a challenging domain for existing ACSs, as it introduces an intermediate level of administration for virtual machines (VM). Moreover, it sometimes requires the use of several clouds at once (hybrid and multi-cloud). To make matter worse, clouds API are not standardized, which results to non interoperable clouds. For example, running a financial/bank application within the cloud generally requires two clouds: a private cloud (located in the company to which the application belongs [1]) to run business-critical part, and a public cloud (e.g. Amazon EC2) to run non-critical part of the application. Note that in some situations, the non-critical part can move from one cloud to another for price and competitiveness reasons. In this context, existing ACSs [3], [4] are inappropriate for several reasons. (1) They only consider one level of deployment/execution: an application runs within a physical machine (PM), whereas in the context of cloud, the application runs within a VM, which in turn runs on a PM. (2) The target execution environment is not static in the context of cloud, an application does not stay within the same cloud during its overall lifetime. (3) Existing ACSs are built to administrate the whole environment (application and execution environment) while in the context of cloud, administration is ensured by two actors (the deployer administrates its application while the cloud provider administrates VMs and PMs).

This paper addresses these problems by proposing Roboconf, a generic (administrate any kind of applications) open source[2], extensible, multi-cloud (target several clouds at once), scalable, and fine-grained reconfigurable orchestration framework. A hybrid cloud orchestrator is a software managing interactions and interconnections among on-premises and cloud-based business units. Cloud orchestrator products connect various automated processes and associated resources using mechanisms like ACS. The key ideas behind Roboconf are the following: (1) Roboconf is a ACS kernel which implements basic administration mechanisms (sensor and effector); (2) the latter are easily improvable by any deployer; (3) Roboconf is provided with a set of reusable components, each of them implements an administrative need; and (4) a hierarchical DSL (Domain Specific Language) for a fine-grained expression of applications and execution environments. In order to keep Roboconf simple, we focus on a subset of administration stacks: initial installation (including initialization, deployment, configuration, start-up, stopping and uninstallation), dynamic installation (at runtime), and incrementally partial (only a part of the application) or full application installation. We have performed several experiments validating all the properties of Roboconf. The rest of the paper is organized as follows. Section II motivates this work. Section III presents Roboconf. Its evaluation and results are presented and discussed in Section IV. Section V reviews the related work. Finally, Section VI concludes the paper.

[1]Called deployer in this paper.

[2]http://roboconf.net/

II. A Motivating Use Case

Let consider a company which wants to enjoy the benefits of cloud computing. This company has an e-commerce application represented here by the RUBiS [22] benchmark. RUBiS is a JEE application based on servlets, which implements an auction web site modelled on eBay. RUBiS defines interactions such as registering new users, browsing, buying or selling items. To run this application, the company's administrator decides to use a web server provided by Apache HTTPD, an application server provided by Tomcat, and a set of database servers provided by MySQL. Apache relies on Mod_JK connector to forward requests to Tomcat, via its AJP 13 connector. Let us consider a scenario where the company has the following requirements. Most of its clients (users who connect to its application) are located on the one hand in France (near Marseille) and on the other hand in Brazil (Sao Paulo). The company organizes data into two categories: business-critical (e.g. those which concern money) and non-critical (e.g. those which concern sold items). The former must be located on company premises, which is composed of a virtualized machine (provided by VMware vSphere) and a native machine. The virtualized machine runs the database which is in production while the native machine runs a backup. Regarding non-critical data, they are hosted (with the other application components) within any public clouds (the most cheapest one which is near company's clients). The company capitalizes on competition among cloud providers and fully benefits from them. According to VM prices charged on clouds market, the company runs its application within two distinct clouds: Amazon EC2 and Microsoft Azure. Concerning the administration of the application, the company's administrator practices a fine-grained administration such as manipulating a .war package in a Tomcat server or a servlet in the .war package which is in the Tomcat container. Furthermore, sometimes he needs to deploy an entire stack or just a part of the stack. For example, in the case of an intrusion in the VM hosting a Tomcat application server, does he need to redeploy the overall stack? If the intrusion is at the .war package level, only the deployment of the corresponding package and its servlets are needed. If the problem comes from a single servlet, only this servlet is taken into account. Another need is the reconfiguration of the application, partly or entirely during its lifetime, especially in moving a portion from a cloud to another one. Figure 1 depicts this scenario. This example depicts a trend as to be shown in the 2014 State of the Cloud Survey, "the hybrid and multi-cloud implementations continue to be the end goal for the enterprise: 74% of enterprise respondents have a multi-cloud strategy, and 48% are planning for hybrid clouds." [11].

In summary, this practical use case intuitively points out the following features from the ACS which attempts to conveniently administrate it. (1) The ACS should be able to provide both hybrid and multi-cloud deployment features, with the target clouds unknown in advance. (2) It should provide a hierarchical language for expressing the use case in order to allow a fine-grained administration. Roboconf, such a system, is described in next section.

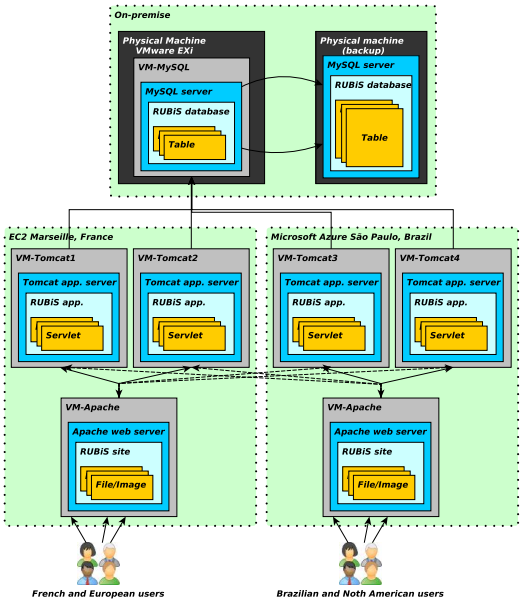

Fig. 1: A multi-cloud deployment of the RUBiS benchmark

III. Roboconf

A. Overview

Roboconf is a distributed solution to deploy distributed applications. It is an open-source software licensed under the terms of the Apache license version 2.0. It is a deployment framework for multi-cloud, but not only. It allows to describe distributed applications and handle deployment automatically of the entire application or a part of it. The objective of this framework is to be improvable with a micro kernel which is the core of Roboconf. This kernel implements all necessary mechanism to plug new behaviours for addressing new applications and new execution environment. Moreover, Roboconf supports scale-up and scale-down natively. Its main force is the support of dynamic (re)configuration. This provides a lot of flexibility and allows elastic deployments. Roboconf is made up of several modules. A simplified drawing of Roboconf architecture is depicted in Figure 2 and explained more detail as follows.

The **Deployment Manager** (or DM) is an application in charge of managing VM and the agents (see below). It acts as an interface to the set of VMs or devices. It is also in charge of instantiating VMs in the IaaS and PMs such as embedded boards. The **Agent** is a Software component that must be deployed on every VM and device on which Roboconf wants to deploy or control something for bootstrapping. Agents use **plug-ins** to delegate the manipulation of software instances. The plug-ins can be life cycle management ones that support different implementation languages or frameworks such as Bash, Puppet, OSGi [17], Java, etc. It also can be a federated PaaS plug-ins such as Heroku [15] driver.

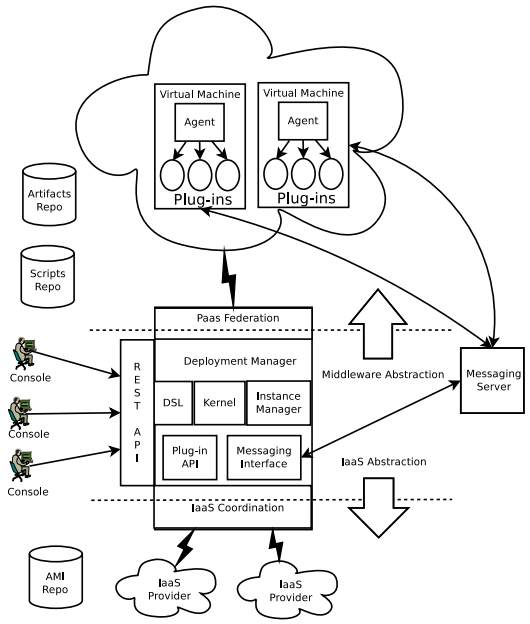

Fig. 2: Simplified architecture of Roboconf

Roboconf's kernel is kept lightweight and the plug-ins can be flexibly plugged into the core. Roboconf agents communicate with each other through an asynchronous messaging server. The **SoftwareInstanceManager** is developed as a Roboconf's plug-in to generate software life-cycle management on different software platform and monitor software instances themselves. The **Messaging Server** is the key component acting as distributed registry of import/export variables that enable communications between the DM and the agents. Roboconf includes the message definitions, the interface to interact with a given messaging server and their implementations. The DM and the agents always communicate asynchronously through this server. The **Artifact and VM Image repositories** are responsible for distribution software packages (i.e. artifact) and VM's image, respectively. Artifact repositories can be managed locally or retrieved from public repositories such as Maven center or NPM. Image repository is a database to map each required VM image of each IaaS to corresponding infrastructure components. The required VM image can be an image available in the VM image marketplace provided by IaaS or a pre-built one created manually or automatically (e.g. using Dockerfile [24] or Vagrantfile [25]). Eventually, an **admin console** is required to control the DM. Roboconf comes with a shell-based console and an AngularJS web application providing different user interfaces to interact with the DM through REST. It contains utilities to transform Java beans into JSON.

Roboconf takes as input the description of a whole application in terms of "components" and "instances". Components can be seen as object definitions, while instances are obviously instances of these objects. From this model, it then takes the burden of launching VMs, deploying software on them, resolving dependencies dynamically among software components, updating their configuration and starting the whole stuff when ready.

Roboconf handles application life cycle: hot reconfiguration (e.g. for elasticity issues) and consistency (e.g. maintaining a consistent state when a component starts or stops, even accidentally). This relies on a messaging queue (currently RabbitMQ [16]). Application parts know what they expose to and what they depend on from other parts. The global idea is to apply to applications the concepts used in component technologies like OSGi. Roboconf achieves this in a non-intrusive way, so that it can work with legacy software. Application parts use the message queue to communicate and take the appropriate actions depending on what is deployed or started. These appropriate actions are executed by common plug-ins such as bash, puppet or customized ones such as java-servlet, osgi-bundle.

Roboconf is a distributed technology, based on AMQP [18] and REST/JSON. It is both IaaS and PaaS-agnostic. Many well-known IaaS are supported including OpenStack, Amazon Web Services, Microsoft Windows Azure, VMware vSphere, a plug-in to deploy Docker container as well as a "local" deployment plug-in for on-premise hosts. In the PaaS aspect, not only potential type of applications are tensely brought up to the Cloud such as OSGi or Internet of Things (i.e. IoT) but also state-of-the-art PaaS are purposefully included such as Heroku, Google App Engine and CloudBees.

Roboconf satisfies most of state-of-the-art requirements of a modern multi-cloud PaaS such as component fine-grained hierarchical description, dynamic dependency resolution, concurrent component deployment, multi-cloud distributed deployment, middleware-orientation, genericity, extensibility, scalability and reusable/configurable deployment plans.

B. An Architecture with Hierarchical Graph

1) Models: Roboconf is designed to see a distributed application as a set of "components", and as a group of "instances" of these components. Let us take as an example the three-tier distributed application "Apache-Tomcat-MySQL". "Apache" is a component, while an installation of Apache on a particular machine is an instance. Another installation of Apache on another machine is another instance. Besides, Roboconf is built to see distributed application as a group of components that each one exchanges a group of simple data between each other. Data can be string or structured data. Components of a distributed application are composed of variables as for example the ip address or the port used. Parts of those variables may be needed by other components of the application, they are named "exported vars", while vars coming from other components of the application are named "imported vars". Moreover, definition of a component can be inherited by definition of another according to object-oriented design. It inherits all import/export vars and default values. For instance, Tomcat component can inherit properties of a generic "Application Server" component.

Now that we have a far view of an application, let us explain more precisely what are its components. In the above

example, Apache in this case simply imports variables coming from Tomcat: the ip and port of application server. As we said earlier, we define elements (component and instance) as having a set of variables, and having exported and imported variables. In this case of Apache there are only imported variables. A sample of Apache component under Roboconf's language could be as the following:

```
Apache { # Apache Load Balancer (a comment)
  installer: puppet;
  imports: Tomcat.portAJP, Tomcat.ip;
}
```

This small portion is made up of several regions. The **installer**: A component property mandatory and designates the Roboconf plug-in that will handle the life cycle of component instances. In this example, we are using "puppet" implementation. The **imports** lists the variables this components need to be resolved before starting. Variable names are separated by commas. They are also prefixed by the component that exports them. As an example, if Tomcat exports the ip variable, then a depending component will import Tomcat.ip. On the other hand, MySQL does not import data from other components, it is the one exporting data which are its ip and port. A definition of MySQL could be as following:

```
MySQL { # MySQL database
  installer: bash;
  exports: ip, port = 3306;
}
```

Here has a minor different from the **exports** which lists the variables this component makes visible to other components. The "ip" is a special variable name whose value will be set dynamically by Roboconf. All the other variables should specify a default value.

In terms of model and configuration files, Roboconf has the following concepts. The **application descriptor** contains meta-information of the application such as name, version qualifier and description. The **graph** is in fact a set of graphs. It defines software components which range broadly from the (virtual) machine, cloud platform to the application package. The graph defines containment relations and runtime relations. Two kind of relations are defined as follows: (1) Containment means a component can be deployed over another one. As an example, a Tomcat server can be deployed over a VM. Or a web application (WAR) can be deployed over a Tomcat server. (2) Runtime relations refer to components that work together. For instance, a web application needs a database. More specifically, it needs the IP address and the port of the database. Generally, this information is hard-coded. Roboconf can instead resolve them at runtime and update components through the configuration or management APIs (e.g. JMX, REST). As an example, Apache, Tomcat and MySQL can be deployed in parallel. Tomcat will be deployed but will not be able to start until it knows where is the database. Once the database is deployed and started, Roboconf will update Tomcat configuration so that it knows where is MySQL. This is what runtime dependencies make possible. If the graph defines relations between components, **instances** represent concrete components. Like a Java class, a Roboconf component is only a definition. It needs to be instantiated to be used. Predefined instances aim at gaining some time when one wants to deploy application parts. As an example, the deployer could have defined a Tomcat component in the graph, and have four instances, one deployed on machine A, and another on machine B and other two on machine C. These would be four instances of the same component. The rules that apply to them are deduced from the graph, but they have their own configuration.

Roboconf is also designed to see an application as hierarchy of components. The main motivation of hierarchy is to allow Roboconf to exactly keep track of where instances are implemented in the system. It helps Roboconf to make right decisions in dynamic deployment as mentioned in the motivating example. A natural example of parent/children relationship of components of an OSGi application is depicted following:

```
# An Azure VM
VM_AZURE {
  installer: iaas;
  children: Karaf;
}
# Karaf: OSGi container
Karaf {
  installer: bash;
  exports: ip, agentID = 1;
  children: Joram, JNDI;
}
```

```
# Joram: OSGi JMS service
Joram {
  installer: osgi-bundle;
  exports: portJR = 16001;
  imports: Karaf.agentID,
           Karaf.ip;
}
# JNDI: naming service
JNDI {
  installer: osgi-bundle;
  exports: portJNDI = 16401;
  imports: Karaf.agentID,
           Karaf.ip;
}
```

There is a new important field: **children** which lists the components that can be instantiated and deployed over this component. In the example above, it means we can deploy Karaf over a VM instance. In turn, Joram and JNDI can be deployed over instances of Karaf. While hierarchical model resolves the containment relations (i.e. vertical relationship) and the export/import variables model responsible for disentangling the runtime relations (i.e. horizontal relationship) amongst components, a bi-color Graph put everything together in a DSL introduced more details in next section. At runtime, the Graph is used to determine what can be instantiated, and how it can be deployed. Software components include the deployment roots (e.g. VMs, devices, remote hosts), databases, application servers and application modules (e.g. WAR, ZIP, etc). They list what the deployers want to deploy or possibly deploy. What is modelled in the graph is really a user choice. Various granularity can be described. It can goes very deeply in the description (Figure 3) or bundle things together such as associating a given WAR with an application server.

2) Configuration Files and fine-grained hierarchical DSL: An application deployed by Roboconf should provide at least three files. The first is a descriptor application file containing the main Roboconf configuration. This file describes the application itself such as name, description, location of the main model files, etc. Second one is an acyclic graph describing both vertical and horizontal relationships between components of

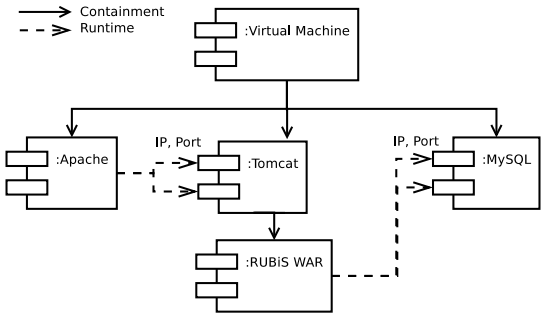

Fig. 3: Illustration of a fine-grained description of components

the application. The components can be software components to install, or VMs to deploy on, etc. Along to the graph file, users also need to provide all resources necessary to deploy the component (e.g. scripts, software packages, configuration files). The IaaS on which the VM will be created are also defined in the component's resources directory. We can choose VMs with pre-defined configuration such as "m1.large" of Amazon EC2 or "Standard A2" of Microsoft Azure. Or we can customize to create dedicated configurations in the case of private clouds or on-premise hosts. On runtime, Roboconf will provision the VMs based on these defined configurations. Figure 4(a) depicts components of the motivating use case described in hierarchical and fine-grained manner using Roboconf's DSL. Final one is an instances file that lists all the initial instances. It means graph components will be pre-instantiated, ready to be deployed. The fact the instances are defined does not mean they will be deployed or started automatically but they will be already defined and configured. In this file, the instances must be defined hierarchically. If the graph defines a root component R with a child C, then an instance of C must be defined in an instance of R. The instance may also declare properties to override component properties. As an example, if a Tomcat component exports a port property with the default value 8080, the instance may override it (e.g. with 8081). An example of this file for the three-tier application is found in Figure 4(b) where an instance of Apache, one instance of MySQL, two instances of Tomcat and one instance of Rubis deployed on different clouds. As far as we know, Roboconf provides a DSL which is inspired from the CSS grammar. It was preferred over XML (easy but heavy), JSON (not user-friendly) and YAML (error prone when many levels of indentation). Its main force is to keep the thing simple, with the minimal set of characters to write. We developed an Eclipse plug-in operating as an editor providing semantic checking and syntax highlighting for the Roboconf's DSL.

So far, the system can understand the distributed application that users want it to deploy. The details about deployment process is discussed in next two subsections.

3) Initial Deployment Process: We use three-tier example to understand the way Roboconf works. As mentioned, dependencies between components is presented in Figure 3. In an IaaS elasticity scenario, multiple Tomcat nodes can be

(a)

```
# An Azure VM
VM_AZURE {
  installer: iaas;
  children: Tomcat,
        Apache, MySQL;
}
# An EC2 VM
VM_EC2 {
  installer: iaas;
  children: Tomcat,
        Apache, MySQL;
}
# A VMware VM
VM_VMWARE {
  installer: iaas;
  children: Tomcat,
        Apache, MySQL;
}
# MySQL
MySQL {
  installer: puppet;
  exports: ip, port = 3306;
}
# Tomcat with Rubis
Tomcat {
  installer: puppet;
  exports: ip,
       portAJP = 8009;
  children: Rubis
}
# Apache Load Balancer
Apache {
  installer: puppet;
  imports: Tomcat.portAJP,
        Tomcat.ip;
}
# RUBiS WAR Application
Rubis {
  installer: java-servlet;
  imports: MySQL.port,
       MySQL.ip;
}
```

(b)

```
# A VM Azure with Apache
instanceof VM_AZURE {
  name: vm-azure-apache;

  instanceof Apache {
    name: apache;
  }
}
# A VM EC2 with Tomcat
instanceof VM_EC2 {
  name: vm-ec2-tomcat-1;

  instanceof Tomcat {
    name: tomcat-1;
    instanceof Rubis {
      name: rubis-1;
    }
  }
}
```

```
# A VM VMware with Tomcat
instanceof VM_VMWARE {
  name: vm-vmware-tomcat-2;

  instanceof Tomcat {
    name: tomcat-2;
  }
}
# A VM VMware with MySQL
instanceof VM_VMWARE {
  name: vm-vmware-mysql;

  instanceof MySQL {
    name: mysql;
  }
}
```

Fig. 4: Example of a Roboconf DSL: (a) Graph and (b) instances file for 3-tier deployment

added/removed to adapt to traffic, but it requires a dynamic reconfiguration of the Apache node in order that mod_proxy knows about all the available Tomcat nodes. Roboconf is told to deploy Apache, MySQL and Tomcat on 3 separate VMs that similar to Figure 4(b). This includes updating the configuration files as soon as dependencies can be resolved (e.g. when it is aware of the MySQL IP/port, Roboconf will send them to the Tomcat node, so it can update its configuration and start). The application components (MySQL, Tomcat, Apache) are defined as Figure 4(a). The VM is supposed to support the deployment of either Apache, Tomcat or MySQL components and each component is described in terms of imports/exports.

With this description, Roboconf knows when a deployed component can be started. It is when all its imports are resolved! Roboconf is in fact responsible for import/export exchanges between components, and life cycle management (e.g. start the component when imports are resolved).

4) Reconfiguration Process: It often happens when everything is running, we need to create a new instance to adapt to changes from environment. It means the running system needs to indicate the component to instantiate, give it a name and define where it should go. In this particular

```
[EVENT nagios peak-load]
GET services
WaitObject: $HOSTNAME CPU load
WaitCondition: CPU load > 80
WaitTrigger: check
```

```
[REACTION peak-load Replicate-Service]
/vmec2/tomcat1
```

Fig. 5: Example of an autonomic rule of Roboconf's DSL: (above) at the agent side, (bottom) at the DM side

example, due to an increasing workload a new Tomcat instance hosted by a VM instance has to be added automatically. We can either reuse an existing or create another VM instance. In this scenario, we take the latter. The Roboconf's DSL provides set of autonomic rules to respond to the detected changes. The agents measure anomalies frequently and send notifications to the DM. The DM responds to the notifications using corresponding imperative rules. Figure 5 depicts a rule that we apply for the example. At the agent side, we use LiveStatus which is the protocol used by Nagios and Shinken. The LiveStatus's query retrieves measures of CPU load from a local Nagios or Shinken agent, if this parameter is over 80%, a notification will be sent to the DM. In turn, the DM applies the handler "Replicate-Service" to respond to the notification resulting in adding an entire new path "/vmec2/tomcat1". Both instances of this path, the "vmec2" and "tomcat1" will be added to the application model. It is one more example emphasizing hierarchy of the Roboconf's DSL.

At the very beginning of the adding process, both the two new are not started, and not even deployed. The DM is asked to deploy and start them. First, the DM provisions the VM. Once it is up, the DM sends the deployment command to the VM and a new Tomcat instance is deployed over it. The Roboconf agents then publishes the exports (i.e. a new Tomcat instance with a port and IP address). Since the Apache load balancer imports such components, it is notified a new Tomcat arrived. The agent associated with the Apache VM invokes a Roboconf plug-in to update the configuration files of the Apache server. Therefore, the load balancer is now aware of two Tomcat servers. If configured in round-robin, it will invoke alternatively every Tomcat server when it receives a request. It is worth noting that real magic with Roboconf is the asynchronous exchange of dependencies between software instances whereas the deployment and life cycle actions are delegated to plug-ins.

IV. Experimental Evaluation

As mentioned in the Subsection III-A, Roboconf provides the following features: component fine-grained hierarchical description, dynamic dependency resolution, concurrent component deployment, multi-cloud distributed deployment, genericity, extensibility, scalability, and dynamic reconfiguration of the deployment plans. To validate those non-functional properties, we conducted a number of experiments with scenarios selected from practical use cases. Concerning the scalability, Roboconf uses the same deployment protocol as [20] which scalability has been demonstrated. We chose different types of application for the experiments to prove the genericity of Roboconf.

TABLE I: Deployment Order of LAMP

Order	Operation	Order	Operation
0	Provision + Boot VM	2	Start - Apache
1	Deploy - Apache	2	Start - MySQL
1	Deploy - MySQL	3	Start - phpMyAdmin
2	Deploy - phpMyAdmin	3	Start - phpMyAdmin

A. Experiment 1

The first type of experiments validates Roboconf in terms of dynamic dependency resolution and concurrent component deployment. To this end, we dissected Roboconf deployment process and compared it with state-of-the-art deployment frameworks: Cloudify, RightScale, and Scalr (that all support concurrent deployment of VMs). Deployment is repeated 8 times for each platform and the means are reported.

a) Scenario and Requirements: For this experiment, we chose EC2 as the target cloud and Puppet as the Roboconf's installer plug-in. We started with a simple LAMP application which is implemented with all-in-one style (Ubuntu 12.04 m3.medium EC2 VM instance). The deployment is considered successful if user can connect and log into phpMyAdmin using any web browsers and start to create a database. The deployment order follows Table I. As mentioned earlier, this experiment was performed on three deployment frameworks. Since each of them has its own states of life cycle, without loss of generality, we distribute those states into two main phases based on classification of [12]:

Booting phase: In this phase deployment systems spend time to process following actions: send "Deploy" requests from client (deployment system client interface) to DM, DM processes the requests, transfer scripts and other necessary files to DM, DM sends requests to IaaS, IaaS provisions and powers up VM, run booting scripts (setup agent, send information of booting machine back to DM).

Operational phase: In this phase the systems consume time to execute operational scripts run once a server is running, on services or components. It may include states: preInstall (download tarball or/and transfer scripts and necessary files to VM, prepare runtime environment, etc), install, postInstall (copy resources and configuration files to right place, set permissions, etc), preStart (resolves dependencies), start, postStart (update variables, configure monitoring), etc.

b) Results: Figure 6 presents results of these experiments. In the operational phase, time is measured until last component is installed. We can see that Roboconf outperforms the others in terms of total deployment time, with Scalr being the nearest one (we only discuss about it). The runner-up, Scalr, took less time in the operational phase than Roboconf because it was not consuming time for dynamic dependency resolution. In Scalr, dependencies amongst components are resolved manually by configuring exchanged variables in its Web UI. In fact, it is a tedious, error-prone and time consuming

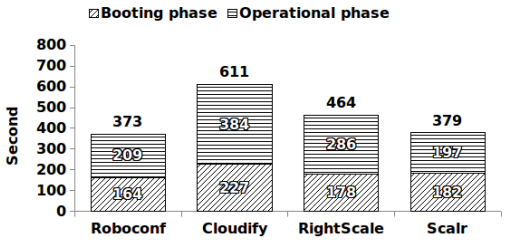

Fig. 6: Deployment time with different deployment systems

job. We also found that in the booting phase, factors that make difference are processing requests and setting up agents, while dependency resolution is mainly diverse element in the operational phase.

B. Experiment 2

The second type of experiments demonstrates the advantage of the fine-grained hierarchical description provided by Roboconf's DSL. For this experiment, EC2 was the target cloud.

a) Scenario and Requirements: We performed this experiment with an OSGi application (the JMS part of the SPECjms2007 benchmark). Regularly, an OSGi application consists of one or several OSGi platforms (e.g. Karaf, Felix, Equinox) providing runtime environment and management framework for OSGi bundles such as Joram, JNDI, etc. We used two instances of EC2 m3.medium, each hosts 2 instances of Karaf. Each Karaf of a VM is customized to choose either Felix or Equinox as underlying OSGi framework and hosts an instance of Joram (an OSGi JMS-supported server), or an instance OSGi JNDI or a JMS OSGi client (publisher/subscriber). Deployment of Joram, JNDI and OSGi JMS clients is handled by the "osgi-bundle" installer, specific to this type of application. We chose Cloudify as comparative objective because it also offers scripting language that can be used to express the structure of a distributed application. Roboconf sees hierarchy of this application whereas a flat structure is seen by Cloudify.

b) Results: With its hierarchical DSL and extensibility, although both solutions need to write 6 sets of configuration scripts for 6 components, Roboconf users only have to write one deployment plan for EC2, one for Karaf and reuse one plan for multiple OSGI bundles (Joram, JNDI, subscriber, publisher). In the case of Cloudify, 6 deployment plans are needed, each one for each component (EC2, Karaf, Joram, JNDI, Subscriber, Publisher). Table II shows statistics about number of the deployment plans for Roboconf and Cloudify, respectively. In this case, Cloudify users have to write twice more deployment plans than Roboconf ones. More details about reusability of Roboconf's DSL can be consulted at [23].

C. Experiment 3

The third type of experiments gives some evidence for the correctness of Roboconf's multi-cloud distributed deployment feature and its extensibility using plug-ins.

TABLE II: Deployment Plans of the OSGi Application

Number of Plan	Roboconf	Cloudify
EC2	1	1
Karaf	1	1
Joram/JNDI/Pub/Sub	1	4
Total	**3**	**6**

a) Scenario and Requirements: We compare deployment time of a Storm [21] cluster (an Event Stream Processing (ESP) application) on multi-cloud platforms using on the one hand of Roboconf and on the other hand a manual configuration following installation guide from original owner. Storm is a part of a global solution for big data analysis. Storm consists of Zookeeper cluster, Nimbus server, Storm supervisors and requires installation of JZMQ, ZeroMQ and Python. The experiment was conducted in a multi-cloud environment combining two public clouds (EC2 and Azure) and a private Cloud (VMware vSphere). Three IaaS plug-ins for these clouds have been developed to provide coordination among the three IaaS providers. Each plug-in needs to implement one Java interface of the Roboconf Plugin API. The LOCs (lines-of-code) for the EC2 plug-in is 202, 393 for Azure, and 157 for VMware vSphere. Zookeeper cluster was installed on EC2s, Nimbus server on Azure and Storm supervisors on our VMware vSphere data-center to take advantage of our computing strength. In this experiment, the time for installing Storm manually is compared with the time to automate its installation using Roboconf.

b) Results: The online installation guide of Storm is 8-page length specific to Storm itself and many external links to resource document of relevant dependencies. One of the authors who had no knowledge about Storm and have never attempted to install this software previously tried to do manual installations. It took him about 6 hours the first time, 3 hours and 30 minutes the second time, and up to 1 hour from the third one. Actions eating effort time were reading imprecise instructions, resolving environment issues, seeking/downloading the required dependencies and debugging problems. On the Roboconf side, the same work has been carried out by another author who also has never known about Storm. With this approach, time mainly devotes for writing deployment plan of Zookeeper, Nimbus, Supervisors, JZMQ, ZeroMQ and Python. About 120 LOC have been written for deploy/start scripts of all Storm's components. After installation, Storm can be managed (deploy, start, stop, undeploy, update) via Roboconf and automatically connect to other applications. At the first time, total development time for Storm in Roboconf was about 2 hours 15 minutes. This time was divided into 30 minutes for component's design, 70 minutes for writing scripts and 35 minutes for debugging and testing. If the required packages are downloaded from the Internet, install of Storm needs 20 minutes and around 7 minutes if the packages are retrieved from a local repository. The automation of the Storm installation via Roboconf empowers Storm developers deploy their existing applications on multi-cloud with slight changes and no need to understand details of Roboconf. It warrants

a repeatable procedure and can be used as a part of larger deployments (e.g. Ubiquitous analytics).

V. Related Work

A number of research in the context of Cloud computing are dedicated either to a single cloud platform or a single type of applications. [6] presents an automatic deployment framework for Xen-based cloud platforms. [7], [9] are description languages for cloud applications. They are limited to three levels of description: PMs, VMs, and applications. Other levels, within the application, are not considered. [8] proposes a language to define the orchestration of the deployment of an application on the cloud. It avoids the description of the target cloud in order to allow the portability of the orchestration. [5] presents some key requirements that a deployment language should respect in order to be used in a multi-cloud deployment system. These requirements take into account the hierarchy of the cloud but does not extend it within the applications as we do in Roboconf. [10] motivates the use of Model Driven Engineering (as we do in Roboconf) to build a useful multi-cloud platform. Therefore, it introduces CloudML which can be seen as a sub-part of Roboconf DSL. Like others, CloudML has no hierarchical structure. In addition, Roboconf does not separate node and artefact types, thus more flexible than CloudML.

Several solutions are close to Roboconf. Ubuntu Juju is able to target a hybrid cloud deployment. However, its description language is not flexible in the sense that it enforces the deployment of a unique component per machine. A hierarchical description is not possible. Docker is a lightweight VM which is aware of applications it runs. Thus it is able to easily deploy and configure. As we do with OSGI containers, Roboconf has been used to automate the setting up of a Docker-containerized application. This plug-in facilitates the continuous integration of both Roboconf test-cases and pre-staging of user deployments and configurations (localhost instead of Cloud tests). Most of them are proprietaries and do not provide detailed documentation about their internal functioning. Cloudify [12] statically provides the possibility to use a number of cloud platforms. Unlike Roboconf, it does not allow to deploy an application within different cloud platforms at the same time. Indeed, it is not able to exchange dependency information or components activation state across different clouds. Therefore, a deployment which requires a private cloud (e.g. financial applications) is not possible with Cloudify. RightScale [13] is another proprietary and commercial solution for deploying applications on the cloud. It proposes a set of applications templates (e.g. JEE) which can be improved by the deployer. Therefore, it does not allow the integration of new templates. No DSL is provided to the deployer. Scalr and EnStratus [14], [19] provide solutions in the same vein as RightScale. In summary, all these solutions are not able to address the use cases we presented in this paper for many reasons.

VI. Conclusion

Roboconf (an open source framework) is a generic, extensible, multi-cloud, scalable, and fine-grained reconfigurable deployment framework. It is based on a lightweight kernel which implements basic administration mechanisms. Its simplicity (regarding its implementation) combined with its component-based approach ease its improvement by any deployer. More important, it provides a hierarchical DSL for a fine-grained expression of applications and execution environments. This allows it to achieve fine-grained level of administration (PM, VM, software within VM, other stacks inside software). The broad variety of experiments we performed validate all its features: multi-cloud, hybrid cloud, LAMP, Storm cluster, etc. In addition, it outperforms (in terms of deployment time and usability) most popular deployment frameworks. A further enhancement to Roboconf would be the implementation of a way to facilitate the integration of more sophisticated reconfiguration policies (e.g. for scalability, fault tolerance).

Acknowledgments

This work is partially supported by the Datalyse FSN project, the FSN OCCIWARE project, the FUI smart support center and thank to Microsoft Research for Windows Azure Grant.

References

[1] Jeffrey O. Kephart and David M. Chess, 'The vision of autonomic computing," Computer 36(1) 2003.
[2] J. O. Kephart, 'Autonomic Computing: The First Decade," ICAC 2011.
[3] Kyle Oppenheim and Patrick McCormick, 'Deployme: Tellme's Package Management and Deployment System," LISA 2000.
[4] E. Yuan, N. Esfahani, and S. Malek, 'A Systematic Survey of Self-Protecting Software Systems," TAAS 2014.
[5] A. Lenk, C. Danschel, M. Klems, D. Bermbach, et al. 'Requirements for an IaaS deployment language in federated Clouds," SOCA 2011.
[6] Y. Zhang, Y. Li, and W. Zheng, 'Automatic software deployment using user-level virtualization for cloud-computing," FGCS 2013.
[7] C. de Alfonso, M. Caballer, F. Alvarruiz, G. Molto, and V. Hernandez, 'Infrastructure deployment over the Cloud," CLOUD 2012.
[8] Tobias Binz, Gerd Breiter, Frank Leymann, and Thomas Spatzier, 'Portable Cloud Services Using TOSCA," IEEE Internet Computing 2012.
[9] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J. Stefani, 'A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures," SPE 2012.
[10] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg, 'Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems," CLOUD 2013.
[11] Cloud Computing Trends: 2014 State of the Cloud Survey, 'http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2014-state-cloud-survey," visited on February 2015.
[12] Cloudify, 'http://www.cloudifysource.org," visited on February 2015.
[13] RightScale, 'www.rightscale.com," visited on February 2015.
[14] Scalr, 'http://www.scalr.com," visited on February 2015.
[15] Heroku, 'https://www.heroku.com," visited on February 2015.
[16] RabbitMQ, 'https://www.rabbitmq.com," visited on February 2015.
[17] OSGI, 'http://www.osgi.org," visited on February 2015.
[18] AMQP, 'http://www.amqp.org," visited on February 2015.
[19] EnStratus, 'http://www.enstratius.com," visited on February 2015.
[20] R. Abid, G. Salaun, F. Bongiovanni, and N. De Palma, 'Verification of a Dynamic Management Protocol for Cloud Applications," ATVA 2013.
[21] Storm, 'http://storm-project.net," visited on February 2015.
[22] RUBiS, 'http://rubis.ow2.org," visited on February 2015.
[23] Roboconf, 'http://roboconf.net/en/user-guide/autonomic-management-with-roboconf.html," visited on February 2015.
[24] Docker, 'https://www.docker.com," visited on February 2015.
[25] Vagrant, 'https://www.vagrantup.com," visited on February 2015.